KernelAnalysis−HOWTO

Table of Contents

Table of Contents

Table of Contents

Table of Contents

KernelAnalysis−HOWTO

Roberto Arcomano berto@bertolinux.com

v0.7, March 26, 2003

This document tries to explain some things about the Linux Kernel, such as the most important components, how they work, and so on. This HOWTO should help prevent the reader from needing to browse all the kernel source files searching for the"right function," declaration, and definition, and then linking each to the other. You can find the latest version of this document at http://www.bertolinux.com If you have suggestions to help make this document better, please submit your ideas to me at the following address: berto@bertolinux.com

---

6. Linux Multitasking

7. Linux Memory Management

8. Linux Networking

9. Linux File System

10. Useful Tips

11. 80386 specific details

12. IRQ

13. Utility functions

14. Static variables

15. Glossary

16. Links

---

1. Introduction

1.1 Introduction

This HOWTO tries to define how parts of the Linux Kernel work, what are the main functions and data structures used, and how the "wheel spins". You can find the latest version of this document at http://www.bertolinux.com If you have suggestions to help make this document better, please submit your ideas to me at the following address: berto@bertolinux.comCode used within this document refers to the Linux Kernel version 2.4.x, which is the last stable kernel version at time of writing this HOWTO.

1.2 Copyright

1.3 Translations

If you want to translate this document you are free to do so. However, you will need to do the following:

1. Check that another version of the document doesn't already exist at your local LDP
2. Maintain all 'Introduction' sections (including 'Introduction', 'Copyright', 'Translations' , 'Credits').

Warning! You don't have to translate TXT or HTML file, you have to modify LYX file, so that it is possible to convert it all other formats (TXT, HTML, RIFF, etc.): to do that you can use "LyX" application you download from http://www.lyx.org.

No need to ask me to translate! You just have to let me know (if you want) about your translation.

Thank you for your translation!

1.4 Credits

Thanks to Linux Documentation Project for publishing and uploading my document quickly.

Thanks to Klaas de Waal for his suggestions.

2. Syntax used

2.1 Function Syntax

When speaking about a function, we write:

```
"function_name  [ file location . extension ]"
```

For example:

```
"schedule [kernel/sched.c]"
```

tells us that we talk about

"schedule"

function retrievable from file

[kernel/sched.c]

Note: We also assume /usr/src/linux as the starting directory.

2.2 Indentation

Indentation in source code is 3 blank characters.

2.3 InterCallings Analysis

Overview

We use the"InterCallings Analysis "(ICA) to see (in an indented fashion) how kernel functions call each other.

For example, the sleep_on command is described in ICA below:

```
|sleep_on
|init_waitqueue_entry      --
|__add_wait_queue            |   enqueuing request
   |list_add                 |
      |__list_add          --
   |schedule              ---     waiting for request to be executed
      |__remove_wait_queue --
      |list_del              |   dequeuing request
         |__list_del       --

                          sleep_on ICA
```

The indented ICA is followed by functions' locations:

- sleep_on [kernel/sched.c]
- init_waitqueue_entry [include/linux/wait.h]
- __add_wait_queue
- list_add [include/linux/list.h]
- __list_add
- schedule [kernel/sched.c]
- __remove_wait_queue [include/linux/wait.h]
- list_del [include/linux/list.h]
- __list_del

Note: We don't specify anymore file location, if specified just before.

Details

In an ICA a line like looks like the following

```
function1 -> function2
```

means that < function1 > is a generic pointer to another function. In this case < function1 > points to < function2 >.

When we write:

```
function:
```

it means that < function > is not a real function. It is a label (typically assembler label).

In many sections we may report a "C" code or a "pseudo−code". In real source files, you could use "assembler" or "not structured" code. This difference is for learning purposes.

PROs of using ICA

The advantages of using ICA (InterCallings Analysis) are many:

- You get an overview of what happens when you call a kernel function
- Function locations are indicated after the function, so ICA could also be considered as a little "function reference"
- InterCallings Analysis (ICA) is useful in sleep/awake mechanisms, where we can view what we do before sleeping, the proper sleeping action, and what we'll do after waking up (after schedule).

CONTROs of using ICA

- Some of the disadvantages of using ICA are listed below:

As all theoretical models, we simplify reality avoiding many details, such as real source code and special conditions.

- Additional diagrams should be added to better represent stack conditions, data values, and so on.

3. Fundamentals

3.1 What is the kernel?

The kernel is the "core" of any computer system: it is the "software" which allows users to share computer resources.

The kernel can be thought as the main software of the OS (Operating System), which may also include graphics management.

For example, under Linux (like other Unix−like OSs), the XWindow environment doesn't belong to the Linux Kernel, because it manages only graphical operations (it uses user mode I/O to access video card devices).

By contrast, Windows environments (Win9x, WinME, WinNT, Win2K, WinXP, and so on) are a mix between a graphical environment and kernel.

3.2 What is the difference between User Mode and Kernel Mode?

Overview

Many years ago, when computers were as big as a room, users ran their applications with much difficulty and, sometimes, their applications crashed the computer.

Operative modes

To avoid having applications that constantly crashed, newer OSs were designed with 2 different operative modes:

1. Kernel Mode: the machine operates with critical data structure, direct hardware (IN/OUT or memory mapped), direct memory, IRQ, DMA, and so on.
2. User Mode: users can run applications.

```
               |          Applications           /|\
               |         ______________           |
               |         | User Mode  |           |
               |         ______________           |
               |               |                  |
Implementation |        _______ _______           |   Abstraction
    Detail     |        | Kernel Mode |           |
               |        _______________           |
               |               |                  |
               |               |                  |
               |               |                  |
              \|/          Hardware               |
```

Kernel Mode "prevents" User Mode applications from damaging the system or its features.

Modern microprocessors implement in hardware at least 2 different states. For example under Intel, 4 states determine the PL (Privilege Level). It is possible to use 0,1,2,3 states, with 0 used in Kernel Mode.

Unix OS requires only 2 privilege levels, and we will use such a paradigm as point of reference.

3.3 Switching from User Mode to Kernel Mode

When do we switch?

Once we understand that there are 2 different modes, we have to know when we switch from one to the other.

Typically, there are 2 points of switching:

1. When calling a System Call: after calling a System Call, the task voluntary calls pieces of code living in Kernel Mode
2. When an IRQ (or exception) comes: after the IRQ an IRQ handler (or exception handler) is called, then control returns back to the task that was interrupted like nothing was happened.

System Calls

System calls are like special functions that manage OS routines which live in Kernel Mode.

A system call can be called when we:

- access an I/O device or a file (like read or write)
- need to access privileged information (like pid, changing scheduling policy or other information)
- need to change execution context (like forking or executing some other application)
- need to execute a particular command (like "chdir", "kill", "brk", or "signal")

```
                                 |                |
                         ------->| System Call i  | (Accessing Devices)
|                |       |       |  [sys_read()]  |
| ...            |       |       |                |
| system_call(i) |--------       |                |
|   [read()]     |               |                |
| ...            |               |                |
| system_call(j) |--------       |                |
|   [get_pid()]  |       |       |                |
| ...            |       ------->| System Call j  | (Accessing kernel data structures)
|                |               |  [sys_getpid()]|
                                 |                |

    USER MODE                        KERNEL MODE


                        Unix System Calls Working
```

System calls are almost the only interface used by User Mode to talk with low level resources (hardware). The only exception to this statement is when a process uses "ioperm" system call. In this case a device can be accessed directly by User Mode process (IRQs cannot be used).

NOTE: Not every "C" function is a system call, only some of them.

Below is a list of System Calls under Linux Kernel 2.4.17, from [arch/i386/kernel/entry.S]

```
.long SYMBOL_NAME(sys_ni_syscall)       /* 0  -  old "setup()" system call*/
.long SYMBOL_NAME(sys_exit)
.long SYMBOL_NAME(sys_fork)
.long SYMBOL_NAME(sys_read)
.long SYMBOL_NAME(sys_write)
.long SYMBOL_NAME(sys_open)             /* 5 */
.long SYMBOL_NAME(sys_close)
.long SYMBOL_NAME(sys_waitpid)
.long SYMBOL_NAME(sys_creat)
.long SYMBOL_NAME(sys_link)
.long SYMBOL_NAME(sys_unlink)           /* 10 */
.long SYMBOL_NAME(sys_execve)
.long SYMBOL_NAME(sys_chdir)
.long SYMBOL_NAME(sys_time)
.long SYMBOL_NAME(sys_mknod)
.long SYMBOL_NAME(sys_chmod)            /* 15 */
.long SYMBOL_NAME(sys_lchown16)
.long SYMBOL_NAME(sys_ni_syscall)                               /* old break syscall holder */
.long SYMBOL_NAME(sys_stat)
.long SYMBOL_NAME(sys_lseek)
.long SYMBOL_NAME(sys_getpid)           /* 20 */
.long SYMBOL_NAME(sys_mount)
.long SYMBOL_NAME(sys_oldumount)
.long SYMBOL_NAME(sys_setuid16)
.long SYMBOL_NAME(sys_getuid16)
.long SYMBOL_NAME(sys_stime)            /* 25 */
.long SYMBOL_NAME(sys_ptrace)
.long SYMBOL_NAME(sys_alarm)
.long SYMBOL_NAME(sys_fstat)
.long SYMBOL_NAME(sys_pause)
.long SYMBOL_NAME(sys_utime)            /* 30 */
.long SYMBOL_NAME(sys_ni_syscall)                               /* old stty syscall holder */
.long SYMBOL_NAME(sys_ni_syscall)                               /* old gtty syscall holder */
.long SYMBOL_NAME(sys_access)
.long SYMBOL_NAME(sys_nice)
.long SYMBOL_NAME(sys_ni_syscall)       /* 35 */                /* old ftime syscall holder */
.long SYMBOL_NAME(sys_sync)
.long SYMBOL_NAME(sys_kill)
.long SYMBOL_NAME(sys_rename)
.long SYMBOL_NAME(sys_mkdir)
.long SYMBOL_NAME(sys_rmdir)            /* 40 */
.long SYMBOL_NAME(sys_dup)
.long SYMBOL_NAME(sys_pipe)
.long SYMBOL_NAME(sys_times)
.long SYMBOL_NAME(sys_ni_syscall)                               /* old prof syscall holder */
.long SYMBOL_NAME(sys_brk)              /* 45 */
.long SYMBOL_NAME(sys_setgid16)
.long SYMBOL_NAME(sys_getgid16)
.long SYMBOL_NAME(sys_signal)
.long SYMBOL_NAME(sys_geteuid16)
.long SYMBOL_NAME(sys_getegid16)        /* 50 */
.long SYMBOL_NAME(sys_acct)
.long SYMBOL_NAME(sys_umount)                                   /* recycled never used phys() */
.long SYMBOL_NAME(sys_ni_syscall)                               /* old lock syscall holder */
.long SYMBOL_NAME(sys_ioctl)
.long SYMBOL_NAME(sys_fcntl)            /* 55 */
.long SYMBOL_NAME(sys_ni_syscall)                               /* old mpx syscall holder */
.long SYMBOL_NAME(sys_setpgid)
.long SYMBOL_NAME(sys_ni_syscall)                               /* old ulimit syscall holder */
.long SYMBOL_NAME(sys_olduname)
.long SYMBOL_NAME(sys_umask)            /* 60 */
.long SYMBOL_NAME(sys_chroot)
```

```
        .long SYMBOL_NAME(sys_ustat)
        .long SYMBOL_NAME(sys_dup2)
        .long SYMBOL_NAME(sys_getppid)
        .long SYMBOL_NAME(sys_getpgrp)          /* 65 */
        .long SYMBOL_NAME(sys_setsid)
        .long SYMBOL_NAME(sys_sigaction)
        .long SYMBOL_NAME(sys_sgetmask)
        .long SYMBOL_NAME(sys_ssetmask)
        .long SYMBOL_NAME(sys_setreuid16)       /* 70 */
        .long SYMBOL_NAME(sys_setregid16)
        .long SYMBOL_NAME(sys_sigsuspend)
        .long SYMBOL_NAME(sys_sigpending)
        .long SYMBOL_NAME(sys_sethostname)
        .long SYMBOL_NAME(sys_setrlimit)        /* 75 */
        .long SYMBOL_NAME(sys_old_getrlimit)
        .long SYMBOL_NAME(sys_getrusage)
        .long SYMBOL_NAME(sys_gettimeofday)
        .long SYMBOL_NAME(sys_settimeofday)
        .long SYMBOL_NAME(sys_getgroups16)      /* 80 */
        .long SYMBOL_NAME(sys_setgroups16)
        .long SYMBOL_NAME(old_select)
        .long SYMBOL_NAME(sys_symlink)
        .long SYMBOL_NAME(sys_lstat)
        .long SYMBOL_NAME(sys_readlink)         /* 85 */
        .long SYMBOL_NAME(sys_uselib)
        .long SYMBOL_NAME(sys_swapon)
        .long SYMBOL_NAME(sys_reboot)
        .long SYMBOL_NAME(old_readdir)
        .long SYMBOL_NAME(old_mmap)             /* 90 */
        .long SYMBOL_NAME(sys_munmap)
        .long SYMBOL_NAME(sys_truncate)
        .long SYMBOL_NAME(sys_ftruncate)
        .long SYMBOL_NAME(sys_fchmod)
        .long SYMBOL_NAME(sys_fchown16)         /* 95 */
        .long SYMBOL_NAME(sys_getpriority)
        .long SYMBOL_NAME(sys_setpriority)
        .long SYMBOL_NAME(sys_ni_syscall)                               /* old profil syscall holder */
        .long SYMBOL_NAME(sys_statfs)
        .long SYMBOL_NAME(sys_fstatfs)          /* 100 */
        .long SYMBOL_NAME(sys_ioperm)
        .long SYMBOL_NAME(sys_socketcall)
        .long SYMBOL_NAME(sys_syslog)
        .long SYMBOL_NAME(sys_setitimer)
        .long SYMBOL_NAME(sys_getitimer)        /* 105 */
        .long SYMBOL_NAME(sys_newstat)
        .long SYMBOL_NAME(sys_newlstat)
        .long SYMBOL_NAME(sys_newfstat)
        .long SYMBOL_NAME(sys_uname)
        .long SYMBOL_NAME(sys_iopl)             /* 110 */
        .long SYMBOL_NAME(sys_vhangup)
        .long SYMBOL_NAME(sys_ni_syscall)       /* old "idle" system call */
        .long SYMBOL_NAME(sys_vm86old)
        .long SYMBOL_NAME(sys_wait4)
        .long SYMBOL_NAME(sys_swapoff)          /* 115 */
        .long SYMBOL_NAME(sys_sysinfo)
        .long SYMBOL_NAME(sys_ipc)
        .long SYMBOL_NAME(sys_fsync)
        .long SYMBOL_NAME(sys_sigreturn)
        .long SYMBOL_NAME(sys_clone)            /* 120 */
        .long SYMBOL_NAME(sys_setdomainname)
        .long SYMBOL_NAME(sys_newuname)
        .long SYMBOL_NAME(sys_modify_ldt)
```

```
.long SYMBOL_NAME(sys_adjtimex)
.long SYMBOL_NAME(sys_mprotect)         /* 125 */
.long SYMBOL_NAME(sys_sigprocmask)
.long SYMBOL_NAME(sys_create_module)
.long SYMBOL_NAME(sys_init_module)
.long SYMBOL_NAME(sys_delete_module)
.long SYMBOL_NAME(sys_get_kernel_syms)  /* 130 */
.long SYMBOL_NAME(sys_quotactl)
.long SYMBOL_NAME(sys_getpgid)
.long SYMBOL_NAME(sys_fchdir)
.long SYMBOL_NAME(sys_bdflush)
.long SYMBOL_NAME(sys_sysfs)            /* 135 */
.long SYMBOL_NAME(sys_personality)
.long SYMBOL_NAME(sys_ni_syscall)       /* for afs_syscall */
.long SYMBOL_NAME(sys_setfsuid16)
.long SYMBOL_NAME(sys_setfsgid16)
.long SYMBOL_NAME(sys_llseek)           /* 140 */
.long SYMBOL_NAME(sys_getdents)
.long SYMBOL_NAME(sys_select)
.long SYMBOL_NAME(sys_flock)
.long SYMBOL_NAME(sys_msync)
.long SYMBOL_NAME(sys_readv)            /* 145 */
.long SYMBOL_NAME(sys_writev)
.long SYMBOL_NAME(sys_getsid)
.long SYMBOL_NAME(sys_fdatasync)
.long SYMBOL_NAME(sys_sysctl)
.long SYMBOL_NAME(sys_mlock)            /* 150 */
.long SYMBOL_NAME(sys_munlock)
.long SYMBOL_NAME(sys_mlockall)
.long SYMBOL_NAME(sys_munlockall)
.long SYMBOL_NAME(sys_sched_setparam)
.long SYMBOL_NAME(sys_sched_getparam)   /* 155 */
.long SYMBOL_NAME(sys_sched_setscheduler)
.long SYMBOL_NAME(sys_sched_getscheduler)
.long SYMBOL_NAME(sys_sched_yield)
.long SYMBOL_NAME(sys_sched_get_priority_max)
.long SYMBOL_NAME(sys_sched_get_priority_min)  /* 160 */
.long SYMBOL_NAME(sys_sched_rr_get_interval)
.long SYMBOL_NAME(sys_nanosleep)
.long SYMBOL_NAME(sys_mremap)
.long SYMBOL_NAME(sys_setresuid16)
.long SYMBOL_NAME(sys_getresuid16)      /* 165 */
.long SYMBOL_NAME(sys_vm86)
.long SYMBOL_NAME(sys_query_module)
.long SYMBOL_NAME(sys_poll)
.long SYMBOL_NAME(sys_nfsservctl)
.long SYMBOL_NAME(sys_setresgid16)      /* 170 */
.long SYMBOL_NAME(sys_getresgid16)
.long SYMBOL_NAME(sys_prctl)
.long SYMBOL_NAME(sys_rt_sigreturn)
.long SYMBOL_NAME(sys_rt_sigaction)
.long SYMBOL_NAME(sys_rt_sigprocmask)   /* 175 */
.long SYMBOL_NAME(sys_rt_sigpending)
.long SYMBOL_NAME(sys_rt_sigtimedwait)
.long SYMBOL_NAME(sys_rt_sigqueueinfo)
.long SYMBOL_NAME(sys_rt_sigsuspend)
.long SYMBOL_NAME(sys_pread)            /* 180 */
.long SYMBOL_NAME(sys_pwrite)
.long SYMBOL_NAME(sys_chown16)
.long SYMBOL_NAME(sys_getcwd)
.long SYMBOL_NAME(sys_capget)
.long SYMBOL_NAME(sys_capset)           /* 185 */
```

```
        .long SYMBOL_NAME(sys_sigaltstack)
        .long SYMBOL_NAME(sys_sendfile)
        .long SYMBOL_NAME(sys_ni_syscall)               /* streams1 */
        .long SYMBOL_NAME(sys_ni_syscall)               /* streams2 */
        .long SYMBOL_NAME(sys_vfork)            /* 190 */
        .long SYMBOL_NAME(sys_getrlimit)
        .long SYMBOL_NAME(sys_mmap2)
        .long SYMBOL_NAME(sys_truncate64)
        .long SYMBOL_NAME(sys_ftruncate64)
        .long SYMBOL_NAME(sys_stat64)           /* 195 */
        .long SYMBOL_NAME(sys_lstat64)
        .long SYMBOL_NAME(sys_fstat64)
        .long SYMBOL_NAME(sys_lchown)
        .long SYMBOL_NAME(sys_getuid)
        .long SYMBOL_NAME(sys_getgid)           /* 200 */
        .long SYMBOL_NAME(sys_geteuid)
        .long SYMBOL_NAME(sys_getegid)
        .long SYMBOL_NAME(sys_setreuid)
        .long SYMBOL_NAME(sys_setregid)
        .long SYMBOL_NAME(sys_getgroups)        /* 205 */
        .long SYMBOL_NAME(sys_setgroups)
        .long SYMBOL_NAME(sys_fchown)
        .long SYMBOL_NAME(sys_setresuid)
        .long SYMBOL_NAME(sys_getresuid)
        .long SYMBOL_NAME(sys_setresgid)        /* 210 */
        .long SYMBOL_NAME(sys_getresgid)
        .long SYMBOL_NAME(sys_chown)
        .long SYMBOL_NAME(sys_setuid)
        .long SYMBOL_NAME(sys_setgid)
        .long SYMBOL_NAME(sys_setfsuid)         /* 215 */
        .long SYMBOL_NAME(sys_setfsgid)
        .long SYMBOL_NAME(sys_pivot_root)
        .long SYMBOL_NAME(sys_mincore)
        .long SYMBOL_NAME(sys_madvise)
        .long SYMBOL_NAME(sys_getdents64)       /* 220 */
        .long SYMBOL_NAME(sys_fcntl64)
        .long SYMBOL_NAME(sys_ni_syscall)       /* reserved for TUX */
        .long SYMBOL_NAME(sys_ni_syscall)       /* Reserved for Security */
        .long SYMBOL_NAME(sys_gettid)
        .long SYMBOL_NAME(sys_readahead)        /* 225 */
```

IRQ Event

When an IRQ comes, the task that is running is interrupted in order to service the IRQ Handler.

After the IRQ is handled, control returns backs exactly to point of interrupt, like nothing happened.

```
              Running Task
             |-----------|          (3)
NORMAL       |   |       | [break execution] IRQ Handler
EXECUTION (1)|   |       |     ------------->|---------|
             |  \|/      |     |             |  does   |
 IRQ (2)---->| ..        |----->             |  some   |
             |   |       |<-----             |  work   |
BACK TO      |   |       |     |             |  ..(4). |
NORMAL    (6)|  \|/      |     <-------------|_________|
```

```
EXECUTION     |____________|   [return to code]
                                        (5)
               USER MODE                            KERNEL MODE

         User−>Kernel Mode Transition caused by IRQ event
```

The numbered steps below refer to the sequence of events in the diagram above:

1. Process is executing
2. IRQ comes while the task is running.
3. Task is interrupted to call an "Interrupt handler".
4. The "Interrupt handler" code is executed.
5. Control returns back to task user mode (as if nothing happened)
6. Process returns back to normal execution

Special interest has the Timer IRQ, coming every TIMER ms to manage:

1. Alarms
2. System and task counters (used by schedule to decide when stop a process or for accounting)
3. Multitasking based on wake up mechanism after TIMESLICE time.

3.4 Multitasking

Mechanism

The key point of modern OSs is the "Task". The Task is an application running in memory sharing all resources (included CPU and Memory) with other Tasks.

This "resource sharing" is managed by the "Multitasking Mechanism". The Multitasking Mechanism switches from one task to another after a "timeslice" time. Users have the "illusion" that they own all resources. We can also imagine a single user scenario, where a user can have the "illusion" of running many tasks at the same time.

To implement this multitasking, the task uses "the state" variable, which can be:

1. READY, ready for execution
2. BLOCKED, waiting for a resource

The task state is managed by its presence in a relative list: READY list and BLOCKED list.

Task Switching

The movement from one task to another is called "Task Switching". many computers have a hardware instruction which automatically performs this operation. Task Switching occurs in the following cases:

1. After Timeslice ends: we need to schedule a "Ready for execution" task and give it access.
2. When a Task has to wait for a device: we need to schedule a new task and switch to it *

* We schedule another task to prevent "Busy Form Waiting", which occurs when we are waiting for a device instead performing other work.

Task Switching is managed by the "Schedule" entity.

```
Timer    |           |
 IRQ     |           |                            Schedule
  |      |           |                     ________________________
  |----->|   Task 1  |<------------------>|(1)Chooses a Ready Task |
  |      |           |                    |(2)Task Switching       |
  |      |___________|                    |________________________|
  |      |           |                               /|\
  |      |           |                                |
  |      |           |                                |
  |      |           |                                |
  |      |           |                                |
  |----->|   Task 2  |<-------------------------------|
  |      |           |                                |
  |      |___________|                                |
  .      .     .     .                                .
  .      .     .     .                                .
  .      .     .     .                                .
  |      |           |                                |
  |      |           |                                |
  ------>|   Task N  |<--------------------------------
         |           |
         |___________|

            Task Switching based on TimeSlice
```

A typical Timeslice for Linux is about 10 ms.

```
 |           |
 |           | Resource    _____________________________
 |   Task 1  |----------->|(1) Enqueue Resource request |
 |           |  Access    |(2)  Mark Task as blocked    |
 |           |            |(3)  Choose a Ready Task     |
 |___________|            |(4)    Task Switching        |
                          |_____________________________|
                                       |
                                       |
 |           |                         |
 |           |                         |
 |   Task 2  |<-------------------------
 |           |
 |           |
 |___________|

     Task Switching based on Waiting for a Resource
```

3.5 Microkernel vs Monolithic OS

Overview

Until now we viewed so called Monolithic OS, but there is also another kind of OS: "Microkernel".

A Microkernel OS uses Tasks, not only for user mode processes, but also as a real kernel manager, like Floppy−Task, HDD−Task, Net−Task and so on. Some examples are Amoeba, and Mach.

PROs and CONTROs of Microkernel OS

PROS:

- OS is simpler to maintain because each Task manages a single kind of operation. So if you want to modify networking, you modify Net−Task (ideally, if it is not needed a structural update).

CONS:

- Performances are worse than Monolithic OS, because you have to add 2*TASK_SWITCH times (the first to enter the specific Task, the second to go out from it).

My personal opinion is that, Microkernels are a good didactic example (like Minix) but they are not "optimal", so not really suitable. Linux uses a few Tasks, called "Kernel Threads" to implement a little microkernel structure (like kswapd, which is used to retrieve memory pages from mass storage). In this case there are no problems with perfomance because swapping is a very slow job.

3.6 Networking

ISO OSI levels

Standard ISO−OSI describes a network architecture with the following levels:

1. Physical level (examples: PPP and Ethernet)
2. Data−link level (examples: PPP and Ethernet)
3. Network level (examples: IP, and X.25)
4. Transport level (examples: TCP, UDP)
5. Session level (SSL)
6. Presentation level (FTP binary−ascii coding)
7. Application level (applications like Netscape)

The first 2 levels listed above are often implemented in hardware. Next levels are in software (or firmware for routers).

Many protocols are used by an OS: one of these is TCP/IP (the most important living on 3−4 levels).

What does the kernel?

The kernel doesn't know anything (only addresses) about first 2 levels of ISO−OSI.

In RX it:

1. Manages handshake with low levels devices (like ethernet card or modem) receiving "frames" from them.
2. Builds TCP/IP "packets" from "frames" (like Ethernet or PPP ones),
3. Convers "packets" in "sockets" passing them to the right application (using port number) or

4. Forwards packets to the right queue

```
frames         packets              sockets
NIC ---------> Kernel ----------> Application
                  |    packets
                  --------------> Forward
                        - RX -
```

In TX stage it:

1. Converts sockets or
2. Queues datas into TCP/IP "packets"
3. Splits "packets" into "frames" (like Ethernet or PPP ones)
4. Sends "frames" using HW drivers

```
sockets       packets                     frames
Application ---------> Kernel ----------> NIC
              packets     /|\
Forward  -------------------
                        - TX -
```

3.7 Virtual Memory

Segmentation

Segmentation is the first method to solve memory allocation problems: it allows you to compile source code without caring where the application will be placed in memory. As a matter of fact, this feature helps applications developers to develop in a independent fashion from the OS e also from the hardware.

```
            |       Stack        |
            |          |         |
            |         \|/        |
            |        Free        |
            |         /|\        |     Segment <---> Process
            |          |         |
            |        Heap        |
            | Data uninitialized |
            |  Data initialized  |
            |       Code         |
            |____________________|

                  Segment
```

We can say that a segment is the logical entity of an application, or the image of the application in memory.

When programming, we don't care where our data is put in memory, we only care about the offset inside our segment (our application).

We use to assign a Segment to each Process and vice versa. In Linux this is not true. Linux uses only 4 segments for either Kernel and all Processes.

Problems of Segmentation

```
                                         ______________________
                                   ----->|                    |----->
                                   | IN  |     Segment A      | OUT
 ______________________            |     |____________________|
|                      |_____|     |                    |
|      Segment B       |           |     Segment B      |
|                      |_____      |                    |
|______________________|     |     |____________________|
                             |     |     Segment C      |
                             |     |____________________|
                       ----->|     Segment D      |----->
                         IN  |____________________| OUT

                    Segmentation problem
```

In the diagram above, we want to get exit processes A, and D and enter process B. As we can see there is enough space for B, but we cannot split it in 2 pieces, so we CANNOT load it (memory out).

The reason this problem occurs is because pure segments are continuous areas (because they are logical areas) and cannot be split.

Pagination

```
             ____________________
            |     Page 1         |
            |____________________|
            |     Page 2         |
            |____________________|
            |      ..            |     Segment <---> Process
            |____________________|
            |     Page n         |
            |____________________|
            |                    |
            |____________________|
            |                    |
            |____________________|

                  Segment
```

Pagination splits memory in "n" pieces, each one with a fixed length.

A process may be loaded in one or more Pages. When memory is freed, all pages are freed (see Segmentation Problem, before).

Pagination is also used for another important purpose, "Swapping". If a page is not present in physical memory then it generates an EXCEPTION, that will make the Kernel search for a new page in storage memory. This mechanism allow OS to load more applications than the ones allowed by physical memory only.

Pagination Problem

```
                  ____________________
   Page   X |       Process Y        |
            |________________________|
            |                        |
            |        WASTE           |
            |        SPACE           |
            |________________________|

               Pagination Problem
```

In the diagram above, we can see what is wrong with the pagination policy: when a Process Y loads into Page X, ALL memory space of the Page is allocated, so the remaining space at the end of Page is wasted.

Segmentation and Pagination

How can we solve segmentation and pagination problems? Using either 2 policies.

```
                                  |         ..          |
                                  |_____________________|
                            ----->|       Page 1        |
                            |     |_____________________|
                            |     |         ..          |
 ____________________       |     |_____________________|
|                    |      |---->|       Page 2        |
|      Segment X     |  ----|     |_____________________|
|                    |      |     |          ..         |
|____________________|      |     |_____________________|
                            |     |          ..         |
                            |     |_____________________|
                            |---->|       Page 3        |
                                  |_____________________|
                                  |          ..         |
```

Process X, identified by Segment X, is split in 3 pieces and each of one is loaded in a page.

We do not have:

1. Segmentation problem: we allocate per Pages, so we also free Pages and we manage free space in an optimized way.
2. Pagination problem: only last page wastes space, but we can decide to use very small pages, for example 4096 bytes length (losing at maximum 4096*N_Tasks bytes) and manage hierarchical paging (using 2 or 3 levels of paging)

```
                          |         |           |          |
                          |         |  Offset2  |  Value   |
                          |         |       /|\|           |
                  Offset1 |         |-----    | |          |
                     /|\  |         |    |    | |          |
                      |   |         |    |   \|/|          |
```

```
                      |  |         |    ------>|         |
                     \|/ |         |           |         |
 Base Paging Address ---->|         |           |         |
                          | ....... |           | ....... |
                          |         |           |         |

                     Hierarchical Paging
```

4. Linux Startup

We start the Linux kernel first from C code executed from "startup_32:" asm label:

```
|startup_32:
   |start_kernel
      |lock_kernel
      |trap_init
      |init_IRQ
      |sched_init
      |softirq_init
      |time_init
      |console_init
      |#ifdef CONFIG_MODULES
         |init_modules
      |#endif
      |kmem_cache_init
      |sti
      |calibrate_delay
      |mem_init
      |kmem_cache_sizes_init
      |pgtable_cache_init
      |fork_init
      |proc_caches_init
      |vfs_caches_init
      |buffer_init
      |page_cache_init
      |signals_init
      |#ifdef CONFIG_PROC_FS
        |proc_root_init
      |#endif
      |#if defined(CONFIG_SYSVIPC)
         |ipc_init
      |#endif
      |check_bugs
      |smp_init
      |rest_init
         |kernel_thread
         |unlock_kernel
         |cpu_idle
```

- startup_32 [arch/i386/kernel/head.S]
- start_kernel [init/main.c]
- lock_kernel [include/asm/smplock.h]
- trap_init [arch/i386/kernel/traps.c]
- init_IRQ [arch/i386/kernel/i8259.c]
- sched_init [kernel/sched.c]
- softirq_init [kernel/softirq.c]
- time_init [arch/i386/kernel/time.c]
- console_init [drivers/char/tty_io.c]

* init_modules [kernel/module.c]
* kmem_cache_init [mm/slab.c]
* sti [include/asm/system.h]
* calibrate_delay [init/main.c]
* mem_init [arch/i386/mm/init.c]
* kmem_cache_sizes_init [mm/slab.c]
* pgtable_cache_init [arch/i386/mm/init.c]
* fork_init [kernel/fork.c]
* proc_caches_init
* vfs_caches_init [fs/dcache.c]
* buffer_init [fs/buffer.c]
* page_cache_init [mm/filemap.c]
* signals_init [kernel/signal.c]
* proc_root_init [fs/proc/root.c]
* ipc_init [ipc/util.c]
* check_bugs [include/asm/bugs.h]
* smp_init [init/main.c]
* rest_init
* kernel_thread [arch/i386/kernel/process.c]
* unlock_kernel [include/asm/smplock.h]
* cpu_idle [arch/i386/kernel/process.c]

The last function "rest_init" does the following:

1. launches the kernel thread "init"
2. calls unlock_kernel
3. makes the kernel run cpu_idle routine, that will be the idle loop executing when nothing is scheduled

In fact the start_kernel procedure never ends. It will execute cpu_idle routine endlessly.

Follows "init" description, which is the first Kernel Thread:

```
|init
   |lock_kernel
   |do_basic_setup
      |mtrr_init
      |sysctl_init
      |pci_init
      |sock_init
      |start_context_thread
      |do_init_calls
         |(*call())-> kswapd_init
   |prepare_namespace
   |free_initmem
   |unlock_kernel
   |execve
```

---

5. Linux Peculiarities

5.1 Overview

Linux has some peculiarities that distinguish it from other OSs. These peculiarities include:

1. Pagination only
2. Softirq
3. Kernel threads
4. Kernel modules
5. "Proc" directory

Flexibility Elements

Points 4 and 5 give system administrators an enormous flexibility on system configuration from user mode allowing them to solve also critical kernel bugs or specific problems without have to reboot the machine. For example, if you needed to change something on a big server and you didn't want to make a reboot, you could prepare the kernel to talk with a module, that you'll write.

5.2 Pagination only

Linux doesn't use segmentation to distinguish Tasks from each other; it uses pagination. (Only 2 segments are used for all Tasks, CODE and DATA/STACK)

We can also say that an interTask page fault never occurs, because each Task uses a set of Page Tables that are different for each Task. There are some cases where different Tasks point to same Page Tables, like shared libraries: this is needed to reduce memory usage; remember that shared libraries are CODE only cause all datas are stored into actual Task stack.

Linux segments

Under the Linux kernel only 4 segments exist:

1. Kernel Code [0x10]
2. Kernel Data / Stack [0x18]
3. User Code [0x23]
4. User Data / Stack [0x2b]

[syntax is "Purpose [Segment]"]

Under Intel architecture, the segment registers used are:

- CS for Code Segment
- DS for Data Segment
- SS for Stack Segment
- ES for Alternative Segment (for example used to make a memory copy between 2 different segments)

So, every Task uses 0x23 for code and 0x2b for data/stack.

Linux pagination

Under Linux 3 levels of pages are used, depending on the architecture. Under Intel only 2 levels are supported. Linux also supports Copy on Write mechanisms (please see Cap.10 for more information).

Why don't interTasks address conflicts exist?

The answer is very very simple: interTask address conflicts cannot exist because they are impossible. Linear −> physical mapping is done by "Pagination", so it just needs to assign physical pages in an univocal fashion.

Do we need to defragment memory?

No. Page assigning is a dynamic process. We need a page only when a Task asks for it, so we choose it from free memory paging in an ordered fashion. When we want to release the page, we only have to add it to the free pages list.

What about Kernel Pages?

Kernel pages have a problem: they can be allocated in a dynamic fashion but we cannot have a guarantee that they are in contiguous area allocation, because linear kernel space is equivalent to physical kernel space.

For Code Segment there is no problem. Boot code is allocated at boot time (so we have a fixed amount of memory to allocate), and on modules we only have to allocate a memory area which could contain module code.

The real problem is the stack segment because each Task uses some kernel stack pages. Stack segments must be contiguous (according to stack definition), so we have to establish a maximum limit for each Task's stack dimension. If we exceed this limit bad things happen. We overwrite kernel mode process data structures.

The structure of the Kernel helps us, because kernel functions are never:

- recursive
- intercalling more than N times.

Once we know N, and we know the average of static variables for all kernel functions, we can estimate a stack limit.

If you want to try the problem out, you can create a module with a function inside calling itself many times. After a fixed number of times, the kernel module will hang because of a page fault exception handler (typically write to a read−only page).

5.3 Softirq

When an IRQ comes, task switching is deferred until later to get better performance. Some Task jobs (that could have to be done just after the IRQ and that could take much CPU in interrupt time, like building up a TCP/IP packet) are queued and will be done at scheduling time (once a time−slice will end).

In recent kernels (2.4.x) the softirq mechanisms are given to a kernel_thread: "ksoftirqd_CPUn". n stands for the number of CPU executing kernel_thread (in a monoprocessor system "ksoftirqd_CPU0" uses PID 3).

Preparing Softirq

Enabling Softirq

"cpu_raise_softirq" is a routine that will wake_up "ksoftirqd_CPU0" kernel thread, to let it manage the enqueued job.

```
|cpu_raise_softirq
   |__cpu_raise_softirq
   |wakeup_softirqd
      |wake_up_process
```

- cpu_raise_softirq [kernel/softirq.c]
- __cpu_raise_softirq [include/linux/interrupt.h]
- wakeup_softirq [kernel/softirq.c]
- wake_up_process [kernel/sched.c]

"__cpu_raise_softirq" routine will set right bit in the vector describing softirq pending.

"wakeup_softirq" uses "wakeup_process" to wake up "ksoftirqd_CPU0" kernel thread.

Executing Softirq

TODO: describing data structures involved in softirq mechanism.

When kernel thread "ksoftirqd_CPU0" has been woken up, it will execute queued jobs

The code of "ksoftirqd_CPU0" is (main endless loop):

```
for (;;) {
   if (!softirq_pending(cpu))
      schedule();
      __set_current_state(TASK_RUNNING);
   while (softirq_pending(cpu)) {
      do_softirq();
      if (current->need_resched)
         schedule
   }
   __set_current_state(TASK_INTERRUPTIBLE)
}
```

- ksoftirqd [kernel/softirq.c]

5.4 Kernel Threads

Even though Linux is a monolithic OS, a few "kernel threads" exist to do housekeeping work.

These Tasks don't utilize USER memory; they share KERNEL memory. They also operate at the highest privilege (RING 0 on a i386 architecture) like any other kernel mode piece of code.

Kernel threads are created by "kernel_thread [arch/i386/kernel/process]" function, which calls "clone" [arch/i386/kernel/process.c] system call from assembler (which is a "fork" like system call):

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
```

```
        long retval, d0;

        __asm__ __volatile__(
                "movl %%esp,%%esi\n\t"
                "int $0x80\n\t"         /* Linux/i386 system call */
                "cmpl %%esp,%%esi\n\t"  /* child or parent? */
                "je 1f\n\t"             /* parent - jump */
                /* Load the argument into eax, and push it.  That way, it does
                 * not matter whether the called function is compiled with
                 * -mregparm or not.  */
                "movl %4,%%eax\n\t"
                "pushl %%eax\n\t"
                "call *%5\n\t"          /* call fn */
                "movl %3,%0\n\t"        /* exit */
                "int $0x80\n"
                "1:\t"
                :"=&a" (retval), "=&S" (d0)
                :"0" (__NR_clone), "i" (__NR_exit),
                 "r" (arg), "r" (fn),
                 "b" (flags | CLONE_VM)
                : "memory");
        return retval;
}
```

Once called, we have a new Task (usually with very low PID number, like 2,3, etc.) waiting for a very slow resource, like swap or usb event. A very slow resource is used because we would have a task switching overhead otherwise.

Below is a list of most common kernel threads (from "ps x" command):

```
PID      COMMAND
 1        init
 2        keventd
 3        kswapd
 4        kreclaimd
 5        bdflush
 6        kupdated
 7        kacpid
67        khubd
```

'init' kernel thread is the first process created, at boot time. It will call all other User Mode Tasks (from file /etc/inittab) like console daemons, tty daemons and network daemons ("rc" scripts).

Example of Kernel Threads: kswapd [mm/vmscan.c].

"kswapd" is created by "clone() [arch/i386/kernel/process.c]"

Initialisation routines:

```
|do_initcalls
   |kswapd_init
      |kernel_thread
         |syscall fork (in assembler)
```

do_initcalls [init/main.c]

kswapd_init [mm/vmscan.c]

kernel_thread [arch/i386/kernel/process.c]

5.5 Kernel Modules

Overview

Linux Kernel modules are pieces of code (examples: fs, net, and hw driver) running in kernel mode that you can add at runtime.

The Linux core cannot be modularized: scheduling and interrupt management or core network, and so on.

Under "/lib/modules/KERNEL_VERSION/" you can find all the modules installed on your system.

Module loading and unloading

To load a module, type the following:

```
insmod MODULE_NAME parameters

example: insmod ne io=0x300 irq=9
```

NOTE: You can use modprobe in place of insmod if you want the kernel automatically search some parameter (for example when using PCI driver, or if you have specified parameter under /etc/conf.modules file).

To unload a module, type the following:

```
 rmmod MODULE_NAME
```

Module definition

A module always contains:

1. "init_module" function, executed at insmod (or modprobe) command
2. "cleanup_module" function, executed at rmmod command

If these functions are not in the module, you need to add 2 macros to specify what functions will act as init and exit module:

1. module_init(FUNCTION_NAME)
2. module_exit(FUNCTION_NAME)

NOTE: a module can "see" a kernel variable only if it has been exported (with macro EXPORT_SYMBOL).

A useful trick for adding flexibility to your kernel

```
// kernel sources side
void (*foo_function_pointer)(void *);

if (foo_function_pointer)
```

```
  (foo_function_pointer)(parameter);



// module side
extern void (*foo_function_pointer)(void *);

void my_function(void *parameter) {
  //My code
}

int init_module() {
  foo_function_pointer = &my_function;
}

int cleanup_module() {
  foo_function_pointer = NULL;
}
```

This simple trick allows you to have very high flexibility in your Kernel, because only when you load the module you'll make "my_function" routine execute. This routine will do everything you want to do: for example "rshaper" module, which controls bandwidth input traffic from the network, works in this kind of matter.

Notice that the whole module mechanism is possible thanks to some global variables exported to modules, such as head list (allowing you to extend the list as much as you want). Typical examples are fs, generic devices (char, block, net, telephony). You have to prepare the kernel to accept your new module; in some cases you have to create an infrastructure (like telephony one, that was recently created) to be as standard as possible.

5.6 Proc directory

Proc fs is located in the /proc directory, which is a special directory allowing you to talk directly with kernel.

Linux uses "proc" directory to support direct kernel communications: this is necessary in many cases, for example when you want see main processes data structures or enable "proxy−arp" feature on one interface and not in others, you want to change max number of threads, or if you want to debug some bus state, like ISA or PCI, to know what cards are installed and what I/O addresses and IRQs are assigned to them.

```
|-- bus
|   |-- pci
|   |   |-- 00
|   |   |   |-- 00.0
|   |   |   |-- 01.0
|   |   |   |-- 07.0
|   |   |   |-- 07.1
|   |   |   |-- 07.2
|   |   |   |-- 07.3
|   |   |   |-- 07.4
|   |   |   |-- 07.5
|   |   |   |-- 09.0
|   |   |   |-- 0a.0
|   |   |   `-- 0f.0
|   |   |-- 01
|   |   |   `-- 00.0
|   |   `-- devices
```

```
|   `-- usb
|-- cmdline
|-- cpuinfo
|-- devices
|-- dma
|-- dri
|   `-- 0
|       |-- bufs
|       |-- clients
|       |-- mem
|       |-- name
|       |-- queues
|       |-- vm
|       `-- vma
|-- driver
|-- execdomains
|-- filesystems
|-- fs
|-- ide
|   |-- drivers
|   |-- hda -> ide0/hda
|   |-- hdc -> ide1/hdc
|   |-- ide0
|   |   |-- channel
|   |   |-- config
|   |   |-- hda
|   |   |   |-- cache
|   |   |   |-- capacity
|   |   |   |-- driver
|   |   |   |-- geometry
|   |   |   |-- identify
|   |   |   |-- media
|   |   |   |-- model
|   |   |   |-- settings
|   |   |   |-- smart_thresholds
|   |   |   `-- smart_values
|   |   |-- mate
|   |   `-- model
|   |-- ide1
|   |   |-- channel
|   |   |-- config
|   |   |-- hdc
|   |   |   |-- capacity
|   |   |   |-- driver
|   |   |   |-- identify
|   |   |   |-- media
|   |   |   |-- model
|   |   |   `-- settings
|   |   |-- mate
|   |   `-- model
|   `-- via
|-- interrupts
|-- iomem
|-- ioports
|-- irq
|   |-- 0
|   |-- 1
|   |-- 10
|   |-- 11
|   |-- 12
|   |-- 13
|   |-- 14
```

```
|   |-- 15
|   |-- 2
|   |-- 3
|   |-- 4
|   |-- 5
|   |-- 6
|   |-- 7
|   |-- 8
|   |-- 9
|   `-- prof_cpu_mask
|-- kcore
|-- kmsg
|-- ksyms
|-- loadavg
|-- locks
|-- meminfo
|-- misc
|-- modules
|-- mounts
|-- mtrr
|-- net
|   |-- arp
|   |-- dev
|   |-- dev_mcast
|   |-- ip_fwchains
|   |-- ip_fwnames
|   |-- ip_masquerade
|   |-- netlink
|   |-- netstat
|   |-- packet
|   |-- psched
|   |-- raw
|   |-- route
|   |-- rt_acct
|   |-- rt_cache
|   |-- rt_cache_stat
|   |-- snmp
|   |-- sockstat
|   |-- softnet_stat
|   |-- tcp
|   |-- udp
|   |-- unix
|   `-- wireless
|-- partitions
|-- pci
|-- scsi
|   |-- ide-scsi
|   |   `-- 0
|   `-- scsi
|-- self -> 2069
|-- slabinfo
|-- stat
|-- swaps
|-- sys
|   |-- abi
|   |   |-- defhandler_coff
|   |   |-- defhandler_elf
|   |   |-- defhandler_lcall7
|   |   |-- defhandler_libcso
|   |   |-- fake_utsname
|   |   `-- trace
|   |-- debug
```

```
|   |-- dev
|   |   |-- cdrom
|   |   |   |-- autoclose
|   |   |   |-- autoeject
|   |   |   |-- check_media
|   |   |   |-- debug
|   |   |   |-- info
|   |   |   `-- lock
|   |   `-- parport
|   |       |-- default
|   |       |   |-- spintime
|   |       |   `-- timeslice
|   |       `-- parport0
|   |           |-- autoprobe
|   |           |-- autoprobe0
|   |           |-- autoprobe1
|   |           |-- autoprobe2
|   |           |-- autoprobe3
|   |           |-- base-addr
|   |           |-- devices
|   |           |   |-- active
|   |           |   `-- lp
|   |           |       `-- timeslice
|   |           |-- dma
|   |           |-- irq
|   |           |-- modes
|   |           `-- spintime
|   |-- fs
|   |   |-- binfmt_misc
|   |   |-- dentry-state
|   |   |-- dir-notify-enable
|   |   |-- dquot-nr
|   |   |-- file-max
|   |   |-- file-nr
|   |   |-- inode-nr
|   |   |-- inode-state
|   |   |-- jbd-debug
|   |   |-- lease-break-time
|   |   |-- leases-enable
|   |   |-- overflowgid
|   |   `-- overflowuid
|   |-- kernel
|   |   |-- acct
|   |   |-- cad_pid
|   |   |-- cap-bound
|   |   |-- core_uses_pid
|   |   |-- ctrl-alt-del
|   |   |-- domainname
|   |   |-- hostname
|   |   |-- modprobe
|   |   |-- msgmax
|   |   |-- msgmnb
|   |   |-- msgmni
|   |   |-- osrelease
|   |   |-- ostype
|   |   |-- overflowgid
|   |   |-- overflowuid
|   |   |-- panic
|   |   |-- printk
|   |   |-- random
|   |   |   |-- boot_id
|   |   |   |-- entropy_avail
```

```
|   |   |   |-- poolsize
|   |   |   |-- read_wakeup_threshold
|   |   |   |-- uuid
|   |   |   `-- write_wakeup_threshold
|   |   |-- rtsig-max
|   |   |-- rtsig-nr
|   |   |-- sem
|   |   |-- shmall
|   |   |-- shmmax
|   |   |-- shmmni
|   |   |-- sysrq
|   |   |-- tainted
|   |   |-- threads-max
|   |   `-- version
|   |-- net
|   |   |-- 802
|   |   |-- core
|   |   |   |-- hot_list_length
|   |   |   |-- lo_cong
|   |   |   |-- message_burst
|   |   |   |-- message_cost
|   |   |   |-- mod_cong
|   |   |   |-- netdev_max_backlog
|   |   |   |-- no_cong
|   |   |   |-- no_cong_thresh
|   |   |   |-- optmem_max
|   |   |   |-- rmem_default
|   |   |   |-- rmem_max
|   |   |   |-- wmem_default
|   |   |   `-- wmem_max
|   |   |-- ethernet
|   |   |-- ipv4
|   |   |   |-- conf
|   |   |   |   |-- all
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- default
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- eth0
```

```
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- eth1
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   `-- lo
|   |   |   |       |-- accept_redirects
|   |   |   |       |-- accept_source_route
|   |   |   |       |-- arp_filter
|   |   |   |       |-- bootp_relay
|   |   |   |       |-- forwarding
|   |   |   |       |-- log_martians
|   |   |   |       |-- mc_forwarding
|   |   |   |       |-- proxy_arp
|   |   |   |       |-- rp_filter
|   |   |   |       |-- secure_redirects
|   |   |   |       |-- send_redirects
|   |   |   |       |-- shared_media
|   |   |   |       `-- tag
|   |   |   |-- icmp_echo_ignore_all
|   |   |   |-- icmp_echo_ignore_broadcasts
|   |   |   |-- icmp_ignore_bogus_error_responses
|   |   |   |-- icmp_ratelimit
|   |   |   |-- icmp_ratemask
|   |   |   |-- inet_peer_gc_maxtime
|   |   |   |-- inet_peer_gc_mintime
|   |   |   |-- inet_peer_maxttl
|   |   |   |-- inet_peer_minttl
|   |   |   |-- inet_peer_threshold
|   |   |   |-- ip_autoconfig
|   |   |   |-- ip_conntrack_max
|   |   |   |-- ip_default_ttl
|   |   |   |-- ip_dynaddr
|   |   |   |-- ip_forward
|   |   |   |-- ip_local_port_range
|   |   |   |-- ip_no_pmtu_disc
|   |   |   |-- ip_nonlocal_bind
|   |   |   |-- ipfrag_high_thresh
|   |   |   |-- ipfrag_low_thresh
|   |   |   |-- ipfrag_time
```

```
|   |   |   |-- neigh
|   |   |   |   |-- default
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_interval
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- gc_thresh1
|   |   |   |   |   |-- gc_thresh2
|   |   |   |   |   |-- gc_thresh3
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   |-- eth0
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   |-- eth1
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   `-- lo
|   |   |   |       |-- anycast_delay
|   |   |   |       |-- app_solicit
|   |   |   |       |-- base_reachable_time
|   |   |   |       |-- delay_first_probe_time
|   |   |   |       |-- gc_stale_time
|   |   |   |       |-- locktime
|   |   |   |       |-- mcast_solicit
|   |   |   |       |-- proxy_delay
|   |   |   |       |-- proxy_qlen
|   |   |   |       |-- retrans_time
|   |   |   |       |-- ucast_solicit
|   |   |   |       `-- unres_qlen
|   |   |   |-- route
|   |   |   |   |-- error_burst
|   |   |   |   |-- error_cost
|   |   |   |   |-- flush
|   |   |   |   |-- gc_elasticity
```

```
|   |   |   |   |-- gc_interval
|   |   |   |   |-- gc_min_interval
|   |   |   |   |-- gc_thresh
|   |   |   |   |-- gc_timeout
|   |   |   |   |-- max_delay
|   |   |   |   |-- max_size
|   |   |   |   |-- min_adv_mss
|   |   |   |   |-- min_delay
|   |   |   |   |-- min_pmtu
|   |   |   |   |-- mtu_expires
|   |   |   |   |-- redirect_load
|   |   |   |   |-- redirect_number
|   |   |   |   `-- redirect_silence
|   |   |   |-- tcp_abort_on_overflow
|   |   |   |-- tcp_adv_win_scale
|   |   |   |-- tcp_app_win
|   |   |   |-- tcp_dsack
|   |   |   |-- tcp_ecn
|   |   |   |-- tcp_fack
|   |   |   |-- tcp_fin_timeout
|   |   |   |-- tcp_keepalive_intvl
|   |   |   |-- tcp_keepalive_probes
|   |   |   |-- tcp_keepalive_time
|   |   |   |-- tcp_max_orphans
|   |   |   |-- tcp_max_syn_backlog
|   |   |   |-- tcp_max_tw_buckets
|   |   |   |-- tcp_mem
|   |   |   |-- tcp_orphan_retries
|   |   |   |-- tcp_reordering
|   |   |   |-- tcp_retrans_collapse
|   |   |   |-- tcp_retries1
|   |   |   |-- tcp_retries2
|   |   |   |-- tcp_rfc1337
|   |   |   |-- tcp_rmem
|   |   |   |-- tcp_sack
|   |   |   |-- tcp_stdurg
|   |   |   |-- tcp_syn_retries
|   |   |   |-- tcp_synack_retries
|   |   |   |-- tcp_syncookies
|   |   |   |-- tcp_timestamps
|   |   |   |-- tcp_tw_recycle
|   |   |   |-- tcp_window_scaling
|   |   |   `-- tcp_wmem
|   |   `-- unix
|   |       `-- max_dgram_qlen
|   |-- proc
|   `-- vm
|       |-- bdflush
|       |-- kswapd
|       |-- max-readahead
|       |-- min-readahead
|       |-- overcommit_memory
|       |-- page-cluster
|       `-- pagetable_cache
|-- sysvipc
|   |-- msg
|   |-- sem
|   `-- shm
|-- tty
|   |-- driver
|   |   `-- serial
|   |-- drivers
```

```
|   |-- ldisc
|   `-- ldiscs
|-- uptime
`-- version
```

In the directory there are also all the tasks using PID as file names (you have access to all Task information, like path of binary file, memory used, and so on).

The interesting point is that you cannot only see kernel values (for example, see info about any task or about network options enabled of your TCP/IP stack) but you are also able to modify some of it, typically that ones under /proc/sys directory:

```
/proc/sys/
          acpi
          dev
          debug
          fs
          proc
          net
          vm
          kernel
```

/proc/sys/kernel

Below are very important and well−know kernel values, ready to be modified:

```
overflowgid
overflowuid
random
threads-max // Max number of threads, typically 16384
sysrq // kernel hack: you can view istant register values and more
sem
msgmnb
msgmni
msgmax
shmmni
shmall
shmmax
rtsig-max
rtsig-nr
modprobe // modprobe file location
printk
ctrl-alt-del
cap-bound
panic
domainname // domain name of your Linux box
hostname // host name of your Linux box
version // date info about kernel compilation
osrelease // kernel version (i.e. 2.4.5)
ostype // Linux!
```

/proc/sys/net

This can be considered the most useful proc subdirectory. It allows you to change very important settings for your network kernel configuration.

```
core
ipv4
ipv6
unix
ethernet
802
```

/proc/sys/net/core

Listed below are general net settings, like "netdev_max_backlog" (typically 300), the length of all your network packets. This value can limit your network bandwidth when receiving packets, Linux has to wait up to scheduling time to flush buffers (due to bottom half mechanism), about 1000/HZ ms

```
  300    *        100             =     30 000
packets     HZ(Timeslice freq)         packets/s

30 000   *       1000             =      30 M
packets     average (Bytes/packet)   throughput Bytes/s
```

If you want to get higher throughput, you need to increase netdev_max_backlog, by typing:

```
echo 4000 > /proc/sys/net/core/netdev_max_backlog
```

Note: Warning for some HZ values: under some architecture (like alpha or arm−tbox) it is 1000, so you can have 300 MBytes/s of average throughput.

/proc/sys/net/ipv4

"ip_forward", enables or disables ip forwarding in your Linux box. This is a generic setting for all devices, you can specify each device you choose.

/proc/sys/net/ipv4/conf/interface

I think this is the most useful /proc entry, because it allows you to change some net settings to support wireless networks (see Wireless−HOWTO for more information).

Here are some examples of when you could use this setting:

- "forwarding", to enable ip forwarding for your interface
- "proxy_arp", to enable proxy arp feature. For more see Proxy arp HOWTO under Linux Documentation Project and Wireless−HOWTO for proxy arp use in Wireless networks.
- "send_redirects" to avoid interface to send ICMP_REDIRECT (as before, see Wireless−HOWTO for more).

---

6. Linux Multitasking

6.1 Overview

This section will analyze data structures−−the mechanism used to manage multitasking environment under Linux.

Task States

A Linux Task can be one of the following states (according to [include/linux.h]):

1. TASK_RUNNING, it means that it is in the "Ready List"
2. TASK_INTERRUPTIBLE, task waiting for a signal or a resource (sleeping)
3. TASK_UNINTERRUPTIBLE, task waiting for a resource (sleeping), it is in same "Wait Queue"
4. TASK_ZOMBIE, task child without father
5. TASK_STOPPED, task being debugged

Graphical Interaction

```
       ______________     CPU Available     ______________
      |              |  ---------------->  |              |
      | TASK_RUNNING |                     | Real Running |
      |______________|  <----------------  |______________|
                           CPU Busy
            |   /|\
Waiting for |    | Resource
 Resource   |    | Available
           \|/   |
    ______________________
   |                      |
   | TASK_INTERRUPTIBLE / |
   | TASK-UNINTERRUPTIBLE |
   |______________________|

                     Main Multitasking Flow
```

6.2 Timeslice

PIT 8253 Programming

Each 10 ms (depending on HZ value) an IRQ0 comes, which helps us in a multitasking environment. This signal comes from PIC 8259 (in arch 386+) which is connected to PIT 8253 with a clock of 1.19318 MHz.

```
    _____         ______        ______
   | CPU |<------| 8259 |------| 8253 |
   |_____| IRQ0  |______|      |___/|\|
                                    |_____ CLK 1.193.180 MHz

// From include/asm/param.h
#ifndef HZ
#define HZ 100
#endif

// From include/asm/timex.h
#define CLOCK_TICK_RATE 1193180 /* Underlying HZ */

// From include/linux/timex.h
#define LATCH ((CLOCK_TICK_RATE + HZ/2) / HZ) /* For divider */

// From arch/i386/kernel/i8259.c
outb_p(0x34,0x43); /* binary, mode 2, LSB/MSB, ch 0 */
outb_p(LATCH & 0xff , 0x40); /* LSB */
outb(LATCH >> 8 , 0x40); /* MSB */
```

So we program 8253 (PIT, Programmable Interval Timer) with LATCH = (1193180/HZ) = 11931.8 when HZ=100 (default). LATCH indicates the frequency divisor factor.

LATCH = 11931.8 gives to 8253 (in output) a frequency of 1193180 / 11931.8 = 100 Hz, so period = 10ms

So Timeslice = 1/HZ.

With each Timeslice we temporarily interrupt current process execution (without task switching), and we do some housekeeping work, after which we'll return back to our previous process.

Linux Timer IRQ ICA

```
Linux Timer IRQ
IRQ 0 [Timer]
  |
\|/
|IRQ0x00_interrupt        //   wrapper IRQ handler
   |SAVE_ALL              ---
      |do_IRQ                |   wrapper routines
         |handle_IRQ_event  ---
            |handler() -> timer_interrupt  // registered IRQ 0 handler
               |do_timer_interrupt
                  |do_timer
                     |jiffies++;
                     |update_process_times
                     |if (--counter <= 0) { // if time slice ended then
                        |counter = 0;        //   reset counter
                        |need_resched = 1;   //   prepare to reschedule
                     |}
         |do_softirq
         |while (need_resched) { // if necessary
            |schedule             //   reschedule
            |handle_softirq
         |}
   |RESTORE_ALL
```

Functions can be found under:

- IRQ0x00_interrupt, SAVE_ALL [include/asm/hw_irq.h]
- do_IRQ, handle_IRQ_event [arch/i386/kernel/irq.c]
- timer_interrupt, do_timer_interrupt [arch/i386/kernel/time.c]
- do_timer, update_process_times [kernel/timer.c]
- do_softirq [kernel/soft_irq.c]
- RESTORE_ALL, while loop [arch/i386/kernel/entry.S]

Notes:

1. Function "IRQ0x00_interrupt" (like others IRQ0xXY_interrupt) is directly pointed by IDT (Interrupt Descriptor Table, similar to Real Mode Interrupt Vector Table, see Cap 11 for more), so EVERY interrupt coming to the processor is managed by "IRQ0x#NR_interrupt" routine, where #NR is the interrupt number. We refer to it as "wrapper irq handler".
2. wrapper routines are executed, like "do_IRQ","handle_IRQ_event" [arch/i386/kernel/irq.c].

3. After this, control is passed to official IRQ routine (pointed by "handler()"), previously registered with "request_irq" [arch/i386/kernel/irq.c], in this case "timer_interrupt" [arch/i386/kernel/time.c].
4. "timer_interrupt" [arch/i386/kernel/time.c] routine is executed and, when it ends,
5. control backs to some assembler routines [arch/i386/kernel/entry.S].

Description:

To manage Multitasking, Linux (like every other Unix) uses a "counter" variable to keep track of how much CPU was used by the task. So, on each IRQ 0, the counter is decremented (point 4) and, when it reaches 0, we need to switch task to manage timesharing (point 4 "need_resched" variable is set to 1, then, in point 5 assembler routines control "need_resched" and call, if needed, "schedule" [kernel/sched.c]).

6.3 Scheduler

The scheduler is the piece of code that chooses what Task has to be executed at a given time.

Any time you need to change running task, select a candidate. Below is the "schedule [kernel/sched.c]" function.

```
|schedule
   |do_softirq // manages post-IRQ work
   |for each task
      |calculate counter
   |prepare_to__switch // does anything
   |switch_mm // change Memory context (change CR3 value)
   |switch_to (assembler)
      |SAVE ESP
      |RESTORE future_ESP
      |SAVE EIP
      |push future_EIP *** push parameter as we did a call
         |jmp __switch_to (it does some TSS work)
         |__switch_to()
          ..
         |ret *** ret from call using future_EIP in place of call address
      new_task
```

6.4 Bottom Half, Task Queues. and Tasklets

Overview

In classic Unix, when an IRQ comes (from a device), Unix makes "task switching" to interrogate the task that requested the device.

To improve performance, Linux can postpone the non−urgent work until later, to better manage high speed event.

This feature is managed since kernel 1.x by the "bottom half" (BH). The irq handler "marks" a bottom half, to be executed later, in scheduling time.

In the latest kernels there is a "task queue"that is more dynamic than BH and there is also a "tasklet" to manage multiprocessor environments.

BH schema is:

1. Declaration
2. Mark
3. Execution

Declaration

```
#define DECLARE_TASK_QUEUE(q) LIST_HEAD(q)
#define LIST_HEAD(name) \
   struct list_head name = LIST_HEAD_INIT(name)
struct list_head {
   struct list_head *next, *prev;
};
#define LIST_HEAD_INIT(name) { &(name), &(name) }

      ''DECLARE_TASK_QUEUE'' [include/linux/tqueue.h, include/linux/list.h]
```

"DECLARE_TASK_QUEUE(q)" macro is used to declare a structure named "q" managing task queue.

Mark

Here is the ICA schema for "mark_bh" [include/linux/interrupt.h] function:

```
|mark_bh(NUMBER)
   |tasklet_hi_schedule(bh_task_vec + NUMBER)
      |insert into tasklet_hi_vec
         |__cpu_raise_softirq(HI_SOFTIRQ)
            |soft_active |= (1 << HI_SOFTIRQ)

                   ''mark_bh''[include/linux/interrupt.h]
```

For example, when an IRQ handler wants to "postpone" some work, it would "mark_bh(NUMBER)", where NUMBER is a BH declarated (see section before).

Execution

We can see this calling from "do_IRQ" [arch/i386/kernel/irq.c] function:

```
|do_softirq
   |h->action(h)-> softirq_vec[TASKLET_SOFTIRQ]->action -> tasklet_action
      |tasklet_vec[0].list->func
```

"h−>action(h);" is the function has been previously queued.

6.5 Very low level routines

set_intr_gate

set_trap_gate

set_task_gate (not used).

(*interrupt)[NR_IRQS](void) = { IRQ0x00_interrupt, IRQ0x01_interrupt, ..}

NR_IRQS = 224 [kernel 2.4.2]

6.6 Task Switching

When does Task switching occur?

Now we'll see how the Linux Kernel switchs from one task to another.

Task Switching is needed in many cases, such as the following:

- when TimeSlice ends, we need to give access to some other task
- when a task decide to access a resource, it sleeps for it, so we have to choose another task
- when a task waits for a pipe, we have to give access to other task, which would write to pipe

Task Switching

```
                           TASK SWITCHING TRICK
#define switch_to(prev,next,last) do {                                  \
        asm volatile("pushl %%esi\n\t"                                  \
                     "pushl %%edi\n\t"                                  \
                     "pushl %%ebp\n\t"                                  \
                     "movl %%esp,%0\n\t"        /* save ESP */          \
                     "movl %3,%%esp\n\t"        /* restore ESP */       \
                     "movl $1f,%1\n\t"          /* save EIP */          \
                     "pushl %4\n\t"             /* restore EIP */       \
                     "jmp __switch_to\n"                                \
                     "1:\t"                                             \
                     "popl %%ebp\n\t"                                   \
                     "popl %%edi\n\t"                                   \
                     "popl %%esi\n\t"                                   \
                     :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
                      "=b" (last)                                       \
                     :"m" (next->thread.esp),"m" (next->thread.eip),    \
                      "a" (prev), "d" (next),                           \
                      "b" (prev));                                      \
} while (0)
```

Trick is here:

1. "pushl %4" which puts future_EIP into the stack
2. "jmp __switch_to" which execute "__switch_to" function, but in opposite of "call" we will return to valued pushed in point 1 (so new Task!)

```
      U S E R   M O D E                K E R N E L    M O D E

 |          |     |          |       |           |     |          |
 |          |     |          | Timer |           |     |          |
 |          |     |  Normal  |  IRQ  |           |     |          |
 |          |     |   Exec   |------>|Timer_Int. |     |          |
 |          |     |     |    |       | ..        |     |          |
 |          |     |    \|/   |       |schedule() |     | Task1 Ret|
```

```
 |          |     |          |        |_switch_to|<--   |  Address |
 |__________|     |__________|        |          |  |  |          |
                                      |          |  |S |          |
Task1 Data/Stack   Task1 Code         |          |  |w |          |
                                      |          |T|i |          |
                                      |          |a|t |          |
 |          |     |          |        |          |s|c |          |
 |          |     |          | Timer  |          |k|h |          |
 |          |     |  Normal  |  IRQ   |          |  |i |          |
 |          |     |   Exec   |------->|Timer_Int.|  |n |          |
 |          |     |     |    |        | ..       |  |g |          |
 |          |     |    \|/   |        |schedule()|  |  | Task2 Ret|
 |          |     |          |        |_switch_to|<--   |  Address |
 |__________|     |__________|        |__________|     |__________|

Task2 Data/Stack   Task2 Code         Kernel Code  Kernel Data/Stack
```

6.7 Fork

Overview

Fork is used to create another task. We start from a Task Parent, and we copy many data structures to Task Child.

```
                      |         |
                      | ..      |
Task Parent           |         |
|         |           |         |
|  fork   |---------->|  CREATE |
|         |          /|   NEW   |
|_________|         / |   TASK  |
                   /  |         |
    ---           /   |         |
    ---          /    | ..      |
                /     |         |
Task Child     /
|         |   /
|  fork   |<-/
|         |
|_________|

              Fork SysCall
```

What is not copied

New Task just created ("Task Child") is almost equal to Parent ("Task Parent"), there are only few differences:

1. obviously PID
2. child "fork()" will return 0, while parent "fork()" will return PID of Task Child, to distinguish them each other in User Mode
3. All child data pages are marked "READ + EXECUTE", no "WRITE" (while parent has WRITE right for its own pages) so, when a write request comes, a "Page Fault" exception is generated which will create a new independent page: this mechanism is called "Copy on Write" (see Cap.10 for more).

Fork ICA

```
|sys_fork
   |do_fork
      |alloc_task_struct
         |__get_free_pages
       |p->state = TASK_UNINTERRUPTIBLE
       |copy_flags
       |p->pid = get_pid
       |copy_files
       |copy_fs
       |copy_sighand
       |copy_mm // should manage CopyOnWrite (I part)
          |allocate_mm
          |mm_init
             |pgd_alloc -> get_pgd_fast
                |get_pgd_slow
          |dup_mmap
             |copy_page_range
                |ptep_set_wrprotect
                   |clear_bit // set page to read-only
          |copy_segments // For LDT
       |copy_thread
          |childregs->eax = 0
          |p->thread.esp = childregs // child fork returns 0
          |p->thread.eip = ret_from_fork // child starts from fork exit
       |retval = p->pid // parent fork returns child pid
       |SET_LINKS // insertion of task into the list pointers
       |nr_threads++ // Global variable
       |wake_up_process(p) // Now we can wake up just created child
       |return retval

               fork ICA
```

- sys_fork [arch/i386/kernel/process.c]
- do_fork [kernel/fork.c]
- alloc_task_struct [include/asm/processor.c]
- __get_free_pages [mm/page_alloc.c]
- get_pid [kernel/fork.c]
- copy_files
- copy_fs
- copy_sighand
- copy_mm
- allocate_mm
- mm_init
- pgd_alloc −> get_pgd_fast [include/asm/pgalloc.h]
- get_pgd_slow
- dup_mmap [kernel/fork.c]
- copy_page_range [mm/memory.c]
- ptep_set_wrprotect [include/asm/pgtable.h]
- clear_bit [include/asm/bitops.h]
- copy_segments [arch/i386/kernel/process.c]
- copy_thread
- SET_LINKS [include/linux/sched.h]
- wake_up_process [kernel/sched.c]

Copy on Write

To implement Copy on Write for Linux:

1. Mark all copied pages as read−only, causing a Page Fault when a Task tries to write to them.
2. Page Fault handler creates a new page.

```
 | Page
 | Fault
 | Exception
 |
 |
 −−−−−−−−−−−> |do_page_fault
                 |handle_mm_fault
                    |handle_pte_fault
                       |do_wp_page
                          |alloc_page      // Allocate a new page
                          |break_cow
                             |copy_cow_page // Copy old page to new one
                             |establish_pte // reconfig Page Table pointers
                                |set_pte

             Page Fault ICA
```

- do_page_fault [arch/i386/mm/fault.c]
- handle_mm_fault [mm/memory.c]
- handle_pte_fault
- do_wp_page
- alloc_page [include/linux/mm.h]
- break_cow [mm/memory.c]
- copy_cow_page
- establish_pte
- set_pte [include/asm/pgtable−3level.h]

---

7. Linux Memory Management

7.1 Overview

Linux uses segmentation + pagination, which simplifies notation.

Segments

Linux uses only 4 segments:

- 2 segments (code and data/stack) for KERNEL SPACE from [0xC000 0000] (3 GB) to [0xFFFF FFFF] (4 GB)
- 2 segments (code and data/stack) for USER SPACE from [0] (0 GB) to [0xBFFF FFFF] (3 GB)

```
                                  __
  4 GB--->|                |     |
          |     Kernel     |     |  Kernel Space (Code + Data/Stack)
```

```
          |                |  __|
  3 GB--->|----------------|  __
          |                |    |
          |                |    |
  2 GB--->|                |    |
          |     Tasks      |    |  User Space (Code + Data/Stack)
          |                |    |
  1 GB--->|                |    |
          |                |    |
          |________________|  __|
0x00000000
         Kernel/User Linear addresses
```

7.2 Specific i386 implementation

Again, Linux implements Pagination using 3 Levels of Paging, but in i386 architecture only 2 of them are really used:

```
   ------------------------------------------------------------------
   L    I    N    E    A    R         A    D    D    R    E    S    S
   ------------------------------------------------------------------
        \___/                 \___/                    \_____/

     PD offset              PF offset                Frame offset
     [10 bits]              [10 bits]                [12 bits]
          |                      |                          |
          |                      |    -----------           |
          |                      |    |  Value  |-----------|---------
          |     |         |      |    |---------|   /|\     |        |
          |     |         |      |    |         |    |      |        |
          |     |         |      |    |         |    | Frame offset  |
          |     |         |      |    |         |   \|/              |
          |     |         |      |    |---------|<------             |
          |     |         |      |    |         |       |            |
          |     |         |      |    |         |       | x 4096     |
          |     |         |  PF offset|_________|-------             |
          |     |         |       /|\ |         |                    |
      PD offset |_________|-----   |  |         |          __________|
            /|\ |         |    |   |  |         |           |
             |  |         |    |  \|/ |         |          \|/
 _____       |  |         |    ------>|_________|   PHYSICAL ADDRESS
|     |     \|/ |         |    x 4096 |         |
| CR3 |--------->|         |           |         |
|_____|         | ....... |           | ....... |
                |         |           |         |

               Page Directory          Page File

                       Linux i386 Paging
```

7.3 Memory Mapping

Linux manages Access Control with Pagination only, so different Tasks will have the same segment addresses, but different CR3 (register used to store Directory Page Address), pointing to different Page Entries.

In User mode a task cannot overcome 3 GB limit (0 x C0 00 00 00), so only the first 768 page directory entries are meaningful (768*4MB = 3GB).

When a Task goes in Kernel Mode (by System call or by IRQ) the other 256 pages directory entries become important, and they point to the same page files as all other Tasks (which are the same as the Kernel).

Note that Kernel (and only kernel) Linear Space is equal to Kernel Physical Space, so:

```
            ________________ _____
           |Other KernelData|___  |  |                |
           |----------------|   |  |__|                |
           |     Kernel     |\  |____|   Real Other   |
  3 GB --->|----------------| \      |   Kernel Data  |
           |                |\ \     |                |
           |              __|_\_\____|__   Real       |
           |      Tasks     |  \ \   |     Tasks      |
           |              __|___\_\__|__   Space      |
           |                |    \ \ |                |
           |                |     \ \|----------------|
           |                |      \ |Real KernelSpace|
           |________________|       \|________________|

           Logical Addresses          Physical Addresses
```

Linear Kernel Space corresponds to Physical Kernel Space translated 3 GB down (in fact page tables are something like { "00000000", "00000001" }, so they operate no virtualization, they only report physical addresses they take from linear ones).

Notice that you'll not have an "addresses conflict" between Kernel and User spaces because we can manage physical addresses with Page Tables.

7.4 Low level memory allocation

Boot Initialization

We start from kmem_cache_init (launched by start_kernel [init/main.c] at boot up).

```
|kmem_cache_init
   |kmem_cache_estimate
```

kmem_cache_init [mm/slab.c]

kmem_cache_estimate

Now we continue with mem_init (also launched by start_kernel[init/main.c])

```
|mem_init
   |free_all_bootmem
```

```
      |free_all_bootmem_core
```

mem_init [arch/i386/mm/init.c]

free_all_bootmem [mm/bootmem.c]

free_all_bootmem_core

Run-time allocation

Under Linux, when we want to allocate memory, for example during "copy_on_write" mechanism (see Cap.10), we call:

```
|copy_mm
   |allocate_mm = kmem_cache_alloc
      |__kmem_cache_alloc
         |kmem_cache_alloc_one
            |alloc_new_slab
               |kmem_cache_grow
                  |kmem_getpages
                     |__get_free_pages
                        |alloc_pages
                           |alloc_pages_pgdat
                              |__alloc_pages
                                 |rmqueue
                                 |reclaim_pages
```

Functions can be found under:

- copy_mm [kernel/fork.c]
- allocate_mm [kernel/fork.c]
- kmem_cache_alloc [mm/slab.c]
- __kmem_cache_alloc
- kmem_cache_alloc_one
- alloc_new_slab
- kmem_cache_grow
- kmem_getpages
- __get_free_pages [mm/page_alloc.c]
- alloc_pages [mm/numa.c]
- alloc_pages_pgdat
- __alloc_pages [mm/page_alloc.c]
- rm_queue
- reclaim_pages [mm/vmscan.c]

TODO: Understand Zones

7.5 Swap

Overview

Swap is managed by the kswapd daemon (kernel thread).

kswapd

As other kernel threads, kswapd has a main loop that wait to wake up.

```
|kswapd
   |// initialization routines
   |for (;;) { // Main loop
      |do_try_to_free_pages
      |recalculate_vm_stats
      |refill_inactive_scan
      |run_task_queue
      |interruptible_sleep_on_timeout // we sleep for a new swap request
   |}
```

* kswapd [mm/vmscan.c]
* do_try_to_free_pages
* recalculate_vm_stats [mm/swap.c]
* refill_inactive_scan [mm/vmswap.c]
* run_task_queue [kernel/softirq.c]
* interruptible_sleep_on_timeout [kernel/sched.c]

When do we need swapping?

Swapping is needed when we have to access a page that is not in physical memory.

Linux uses "kswapd" kernel thread to carry out this purpose. When the Task receives a page fault exception we do the following:

```
 | Page Fault Exception
 | cause by all these conditions:
 |   a-) User page
 |   b-) Read or write access
 |   c-) Page not present
 |
 |
 -----------> |do_page_fault
                 |handle_mm_fault
                    |pte_alloc
                       |pte_alloc_one
                          |__get_free_page = __get_free_pages
                             |alloc_pages
                                |alloc_pages_pgdat
                                   |__alloc_pages
                                      |wakeup_kswapd // We wake up kernel thread kswapd

                  Page Fault ICA
```

* do_page_fault [arch/i386/mm/fault.c]
* handle_mm_fault [mm/memory.c]
* pte_alloc
* pte_alloc_one [include/asm/pgalloc.h]
* __get_free_page [include/linux/mm.h]
* __get_free_pages [mm/page_alloc.c]

* alloc_pages [mm/numa.c]
* alloc_pages_pgdat
* __alloc_pages
* wakeup_kswapd [mm/vmscan.c]

---

8. Linux Networking

8.1 How Linux networking is managed?

There exists a device driver for each kind of NIC. Inside it, Linux will ALWAYS call a standard high level routing: "netif_rx [net/core/dev.c]", which will controls what 3 level protocol the frame belong to, and it will call the right 3 level function (so we'll use a pointer to the function to determine which is right).

8.2 TCP example

We'll see now an example of what happens when we send a TCP packet to Linux, starting from "netif_rx [net/core/dev.c]" call.

Interrupt management: "netif_rx"

```
|netif_rx
   |__skb_queue_tail
      |qlen++
      |* simple pointer insertion *
   |cpu_raise_softirq
      |softirq_active(cpu) |= (1 << NET_RX_SOFTIRQ) // set bit NET_RX_SOFTIRQ in the BH vector
```

Functions:

* __skb_queue_tail [include/linux/skbuff.h]
* cpu_raise_softirq [kernel/softirq.c]

Post Interrupt management: "net_rx_action"

Once IRQ interaction is ended, we need to follow the next part of the frame life and examine what NET_RX_SOFTIRQ does.

We will next call "net_rx_action [net/core/dev.c]" according to "net_dev_init [net/core/dev.c]".

```
|net_rx_action
   |skb = __skb_dequeue (the exact opposite of __skb_queue_tail)
   |for (ptype = first_protocol; ptype < max_protocol; ptype++) // Determine
      |if (skb->protocol == ptype)                               // what is the network protocol
         |ptype->func -> ip_rcv // according to ''struct ip_packet_type [net/ipv4/ip_output.c]''

    **** NOW WE KNOW THAT PACKET IS IP ****
         |ip_rcv
            |NF_HOOK (ip_rcv_finish)
               |ip_route_input // search from routing table to determine function to call
                  |skb->dst->input -> ip_local_deliver // according to previous routing table check, destination is local machine
                     |ip_defrag // reassembles IP fragments
```

```
                    |NF_HOOK (ip_local_deliver_finish)
                       |ipprot->handler -> tcp_v4_rcv // according to ''tcp_protocol [include/net/protocol.c]''

**** NOW WE KNOW THAT PACKET IS TCP ****
                       |tcp_v4_rcv
                          |sk = __tcp_v4_lookup
                          |tcp_v4_do_rcv
                             |switch(sk->state)

*** Packet can be sent to the task which uses relative socket ***
                             |case TCP_ESTABLISHED:
                                |tcp_rcv_established
                                   |__skb_queue_tail // enqueue packet to socket
                                   |sk->data_ready -> sock_def_readable
                                      |wake_up_interruptible


*** Packet has still to be handshaked by 3-way TCP handshake ***
                             |case TCP_LISTEN:
                                |tcp_v4_hnd_req
                                   |tcp_v4_search_req
                                   |tcp_check_req
                                      |syn_recv_sock -> tcp_v4_syn_recv_sock
                                   |__tcp_v4_lookup_established
                             |tcp_rcv_state_process

               *** 3-Way TCP Handshake ***
                                |switch(sk->state)
                                |case TCP_LISTEN: // We received SYN
                                   |conn_request -> tcp_v4_conn_request
                                      |tcp_v4_send_synack // Send SYN + ACK
                                         |tcp_v4_synq_add // set SYN state
                                |case TCP_SYN_SENT: // we received SYN + ACK
                                   |tcp_rcv_synsent_state_process
                                      tcp_set_state(TCP_ESTABLISHED)
                                         |tcp_send_ack
                                            |tcp_transmit_skb
                                               |queue_xmit -> ip_queue_xmit
                                                  |ip_queue_xmit2
                                                     |skb->dst->output
                                |case TCP_SYN_RECV: // We received ACK
                                   |if (ACK)
                                      |tcp_set_state(TCP_ESTABLISHED)
```

Functions can be found under:

- net_rx_action [net/core/dev.c]
- __skb_dequeue [include/linux/skbuff.h]
- ip_rcv [net/ipv4/ip_input.c]
- NF_HOOK −> nf_hook_slow [net/core/netfilter.c]
- ip_rcv_finish [net/ipv4/ip_input.c]
- ip_route_input [net/ipv4/route.c]
- ip_local_deliver [net/ipv4/ip_input.c]
- ip_defrag [net/ipv4/ip_fragment.c]
- ip_local_deliver_finish [net/ipv4/ip_input.c]
- tcp_v4_rcv [net/ipv4/tcp_ipv4.c]
- __tcp_v4_lookup
- tcp_v4_do_rcv

- tcp_rcv_established [net/ipv4/tcp_input.c]
- __skb_queue_tail [include/linux/skbuff.h]
- sock_def_readable [net/core/sock.c]
- wake_up_interruptible [include/linux/sched.h]
- tcp_v4_hnd_req [net/ipv4/tcp_ipv4.c]
- tcp_v4_search_req
- tcp_check_req
- tcp_v4_syn_recv_sock
- __tcp_v4_lookup_established
- tcp_rcv_state_process [net/ipv4/tcp_input.c]
- tcp_v4_conn_request [net/ipv4/tcp_ipv4.c]
- tcp_v4_send_synack
- tcp_v4_synq_add
- tcp_rcv_synsent_state_process [net/ipv4/tcp_input.c]
- tcp_set_state [include/net/tcp.h]
- tcp_send_ack [net/ipv4/tcp_output.c]

Description:

- First we determine protocol type (IP, then TCP)
- NF_HOOK (function) is a wrapper routine that first manages the network filter (for example firewall), then it calls "function".
- After we manage 3−way TCP Handshake which consists of:

```
SERVER (LISTENING)                       CLIENT (CONNECTING)
                           SYN
                   <−−−−−−−−−−−−−−−−−−−


                        SYN + ACK
                   −−−−−−−−−−−−−−−−−−−>


                           ACK
                   <−−−−−−−−−−−−−−−−−−−

                    3−Way TCP handshake
```

- In the end we only have to launch "tcp_rcv_established [net/ipv4/tcp_input.c]" which gives the packet to the user socket and wakes it up.

9. Linux File System

TODO

10. Useful Tips

10.1 Stack and Heap

Overview

Here we view how "stack" and "heap" are allocated in memory

Memory allocation

```
FF..        |                 | <-- bottom of the stack
       /|\  |                 |   |
 higher |   |                 |   |   stack
 values |   |                 |  \|/  growing
            |                 |
XX..        |                 | <-- top of the stack [Stack Pointer]
            |                 |
            |                 |
            |                 |
00..        |_________________| <-- end of stack [Stack Segment]

                   Stack
```

Memory address values start from 00.. (which is also where Stack Segment begins) and they grow going toward FF.. value.

XX.. is the actual value of the Stack Pointer.

Stack is used by functions for:

1. global variables
2. local variables
3. return address

For example, for a classical function:

```
 |int foo_function (parameter_1, parameter_2, ..., parameter_n) {
    |variable_1 declaration;
    |variable_2 declaration;
      ..
    |variable_n declaration;

    |// Body function
    |dynamic variable_1 declaration;
    |dynamic variable_2 declaration;
     ..
    |dynamic variable_n declaration;

    |// Code is inside Code Segment, not Data/Stack segment!

    |return (ret-type) value; // often it is inside some register, for i386 eax register is used.
 |}
we have

          |                       |
          | 1. parameter_1 pushed | \
    S     | 2. parameter_2 pushed |  | Before
    T     | ...................   |  | the calling
```

```
    A     | n. parameter_n pushed | /
    C     | ** Return address **  | -- Calling
    K     | 1. local variable_1   | \
          | 2. local variable_2   |  | After
          | .................     |  | the calling
          | n. local variable_n   | /
          |                       |
         ...                     ...   Free
         ...                     ...   stack
          |                       |
    H     | n. dynamic variable_n | \
    E     | ...................   |  | Allocated by
    A     | 2. dynamic variable_2 |  | malloc & kmalloc
    P     | 1. dynamic variable_1 | /
          |_______________________|

            Typical stack usage

Note: variables order can be different depending on hardware architecture.
```

10.2 Application vs Process

Base definition

We have to distinguish 2 concepts:

- Application: that is the useful code we want to execute
- Process: that is the IMAGE on memory of the application (it depends on memory strategy used, segmentation and/or Pagination).

Often Process is also called Task or Thread.

10.3 Locks

Overview

2 kind of locks:

1. intraCPU
2. interCPU

10.4 Copy_on_write

Copy_on_write is a mechanism used to reduce memory usage. It postpones memory allocation until the memory is really needed.

For example, when a task executes the "fork()" system call (to create another task), we still use the same memory pages as the parent, in read only mode. When a task WRITES into the page, it causes an exception and the page is copied and marked "rw" (read, write).

```
1-) Page X is shared between Task Parent and Task Child
```

```
 Task Parent
 |         | RO Access  ______
 |         |---------->|Page X|
 |_________|           |______|
                          /|\
                           |
 Task Child                |
 |         | RO Access     |
 |         |----------------
 |_________|


2−) Write request
 Task Parent
 |         | RO Access  ______
 |         |---------->|Page X|    Trying to write
 |_________|           |______|
                          /|\
                           |
 Task Child                |
 |         | RO Access     |
 |         |----------------
 |_________|


3−) Final Configuration: Either Task Parent and Task Child have an independent copy of the Page, X and Y
 Task Parent
 |         | RW Access  ______
 |         |---------->|Page X|
 |_________|           |______|


 Task Child
 |         | RW Access  ______
 |         |---------->|Page Y|
 |_________|           |______|
```

11. 80386 specific details

11.1 Boot procedure

```
bbootsect.s [arch/i386/boot]
setup.S (+video.S)
head.S (+misc.c) [arch/i386/boot/compressed]
start_kernel [init/main.c]
```

11.2 80386 (and more) Descriptors

Overview

Descriptors are data structure used by Intel microprocessor i386+ to virtualize memory.

Kind of descriptors

- GDT (Global Descriptor Table)
- LDT (Local Descriptor Table)

* IDT (Interrupt Descriptor Table)

12. IRQ

12.1 Overview

IRQ is an asyncronous signal sent to microprocessor to advertise a requested work is completed

12.2 Interaction schema

```
                                 |<-->  IRQ(0) [Timer]
                                 |<-->  IRQ(1) [Device 1]
                                 | ..
                                 |<-->  IRQ(n) [Device n]
    _____________________________|
     /|\      /|\          /|\
      |        |            |
     \|/      \|/          \|/

    Task(1)  Task(2) ..   Task(N)


             IRQ - Tasks Interaction Schema
```

What happens?

A typical O.S. uses many IRQ signals to interrupt normal process execution and does some housekeeping work. So:

1. IRQ (i) occurs and Task(j) is interrupted
2. IRQ(i)_handler is executed
3. control backs to Task(j) interrupted

Under Linux, when an IRQ comes, first the IRQ wrapper routine (named "interrupt0x??") is called, then the "official" IRQ(i)_handler will be executed. This allows some duties like timeslice preemption.

13. Utility functions

13.1 list_entry [include/linux/list.h]

Definition:

```
#define list_entry(ptr, type, member) \
((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

Meaning:

"list_entry" macro is used to retrieve a parent struct pointer, by using only one of internal struct pointer.

Example:

```
struct __wait_queue {
   unsigned int flags;
   struct task_struct * task;
   struct list_head task_list;
};
struct list_head {
   struct list_head *next, *prev;
};

// and with type definition:
typedef struct __wait_queue wait_queue_t;

// we'll have
wait_queue_t *out list_entry(tmp, wait_queue_t, task_list);

// where tmp point to list_head
```

So, in this case, by means of *tmp pointer [list_head] we retrieve an *out pointer [wait_queue_t].

```
 ____________ <---- *out [we calculate that]
|flags       |             /|\
|task *-->   |              |
|task_list   |<----    list_entry
|  prev * -->|    |         |
|  next * -->|    |         |
|____________|    ----- *tmp [we have this]
```

13.2 Sleep

Sleep code

Files:

- kernel/sched.c
- include/linux/sched.h
- include/linux/wait.h
- include/linux/list.h

Functions:

- interruptible_sleep_on
- interruptible_sleep_on_timeout
- sleep_on
- sleep_on_timeout

Called functions:

- init_waitqueue_entry
- __add_wait_queue
- list_add

* __list_add
* __remove_wait_queue

InterCallings Analysis:

```
|sleep_on
   |init_waitqueue_entry  --
   |__add_wait_queue        |   enqueuing request to resource list
      |list_add              |
         |__list_add        --
   |schedule              ---     waiting for request to be executed
      |__remove_wait_queue --
      |list_del              |   dequeuing request from resource list
         |__list_del        --
```

Description:

Under Linux each resource (ideally an object shared between many users and many processes), , has a queue to manage ALL tasks requesting it.

This queue is called "wait queue" and it consists of many items we'll call the"wait queue element":

```
***   wait queue structure [include/linux/wait.h]  ***


struct __wait_queue {
   unsigned int flags;
   struct task_struct * task;
   struct list_head task_list;
}
struct list_head {
   struct list_head *next, *prev;
};
```

Graphic working:

```
        ***  wait queue element  ***

                             /|\
                              |
       <--[prev *, flags, task *, next *]-->




                 ***  wait queue list ***

          /|\           /|\           /|\                /|\
           |             |             |                  |
--> <--[task1]--> <--[task2]--> <--[task3]--> .... <--[taskN]--> <--
|                                                                  |
|__________________________________________________________________|



              ***   wait queue head ***
```

```
       task1 <--[prev *, lock, next *]--> taskN
```

"wait queue head" point to first (with next *) and last (with prev *) elements of the "wait queue list".

When a new element has to be added, "__add_wait_queue" [include/linux/wait.h] is called, after which the generic routine "list_add" [include/linux/wait.h], will be executed:

```
***   function list_add [include/linux/list.h]  ***

// classic double link list insert
static __inline__ void __list_add (struct list_head * new,  \
                                   struct list_head * prev, \
                                   struct list_head * next) {
   next->prev = new;
   new->next = next;
   new->prev = prev;
   prev->next = new;
}
```

To complete the description, we see also "__list_del" [include/linux/list.h] function called by "list_del" [include/linux/list.h] inside "remove_wait_queue" [include/linux/wait.h]:

```
***   function list_del [include/linux/list.h]  ***


// classic double link list delete
static __inline__ void __list_del (struct list_head * prev, struct list_head * next) {
   next->prev = prev;
   prev->next = next;
}
```

Stack consideration

A typical list (or queue) is usually managed allocating it into the Heap (see Cap.10 for Heap and Stack definition and about where variables are allocated). Otherwise here, we statically allocate Wait Queue data in a local variable (Stack), then function is interrupted by scheduling, in the end, (returning from scheduling) we'll erase local variable.

```
  new task <----|          task1 <------|          task2 <------|
                |                       |                       |
                |                       |                       |
|..........|    |        |..........|   |        |..........|   |
|wait.flags|    |        |wait.flags|   |        |wait.flags|   |
|wait.task_|____|        |wait.task_|___|        |wait.task_|___|
|wait.prev |-->          |wait.prev |-->         |wait.prev |-->
|wait.next |-->          |wait.next |-->         |wait.next |-->
|..        |             |..        |            |..        |
|schedule()|             |schedule()|            |schedule()|
|..........|             |..........|            |..........|
|__________|             |__________|            |__________|

   Stack                    Stack                   Stack
```

14. Static variables

14.1 Overview

Linux is written in "C" language, and as every application has:

1. Local variables
2. Module variables (inside the source file and relative only to that module)
3. Global/Static variables present in only 1 copy (the same for all modules)

When a Static variable is modified by a module, all other modules will see the new value.

Static variables under Linux are very important, cause they are the only kind to add new support to kernel: they typically are pointers to the head of a list of registered elements, which can be:

* added
* deleted
* maybe modified

```
                           _______      _______      _______
Global variable  -------> |Item(1)| -> |Item(2)| -> |Item(3)|  ..
                          |_______|    |_______|    |_______|
```

14.2 Main variables

Current

```
                           _______________
Current ----------------> | Actual process |
                          |________________|
```

Current points to "task_struct" structure, which contains all data about a process like:

* pid, name, state, counter, policy of scheduling
* pointers to many data structures like: files, vfs, other processes, signals...

Current is not a real variable, it is

```
static inline struct task_struct * get_current(void) {
   struct task_struct *current;
   __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
   return current;
}
#define current get_current()
```

Above lines just takes value of "esp" register (stack pointer) and get it available like a variable, from which we can point to our task_struct structure.

From "current" element we can access directly to any other process (ready, stopped or in any other state) kernel data structure, for example changing STATE (like a I/O driver does), PID, presence in ready list or blocked list, etc.

Registered filesystems

```
                          ______      _______      ______
file_systems  ------> | ext2 | -> | msdos | -> | ntfs |
 [fs/super.c]         |______|    |_______|    |______|
```

When you use command like "modprobe some_fs" you will add a new entry to file systems list, while removing it (by using "rmmod") will delete it.

Mounted filesystems

```
                           ______      _______      ______
mount_hash_table  ---->|   /  | -> | /usr  | -> | /var |
[fs/namespace.c]       |______|    |_______|    |______|
```

When you use "mount" command to add a fs, the new entry will be inserted in the list, while an "umount" command will delete the entry.

Registered Network Packet Type

```
                           ______      _______      ______
     ptype_all  ------>|  ip  | -> |  x25  | -> | ipv6 |
[net/core/dev.c]       |______|    |_______|    |______|
```

For example, if you add support for IPv6 (loading relative module) a new entry will be added in the list.

Registered Network Internet Protocol

```
                             ______      _______      _______
inet_protocol_base ----->| icmp | -> |  tcp  | -> |  udp  |
[net/ipv4/protocol.c]    |______|    |_______|    |_______|
```

Also others packet type have many internal protocols in each list (like IPv6).

```
                             ______      _______      _______
inet6_protos ----------->|icmpv6| -> | tcpv6 | -> | udpv6 |
[net/ipv6/protocol.c]    |______|    |_______|    |_______|
```

Registered Network Device

```
                             ______      _______      _______
dev_base --------------->|  lo  | -> |  eth0 | -> |  ppp0 |
[drivers/core/Space.c]   |______|    |_______|    |_______|
```

Registered Char Device

```
                             ______      _______      ________
chrdevs ---------------->|  lp  | -> | keyb  | -> | serial |
[fs/devices.c]           |______|    |_______|    |________|
```

"chrdevs" is not a pointer to a real list, but it is a standard vector.

Registered Block Device

```
                          _______    _______    ________
bdev_hashtable --------->|  fd   | ->|  hd   | ->|  scsi  |
[fs/block_dev.c]         |_______|   |_______|   |________|
```

"bdev_hashtable" is an hash vector.

15. Glossary

16. Links

Official Linux kernels and patches download site

Great documentation about Linux Kernel

Official Kernel Mailing list

Linux Documentation Project Guides